Internet Programming Series

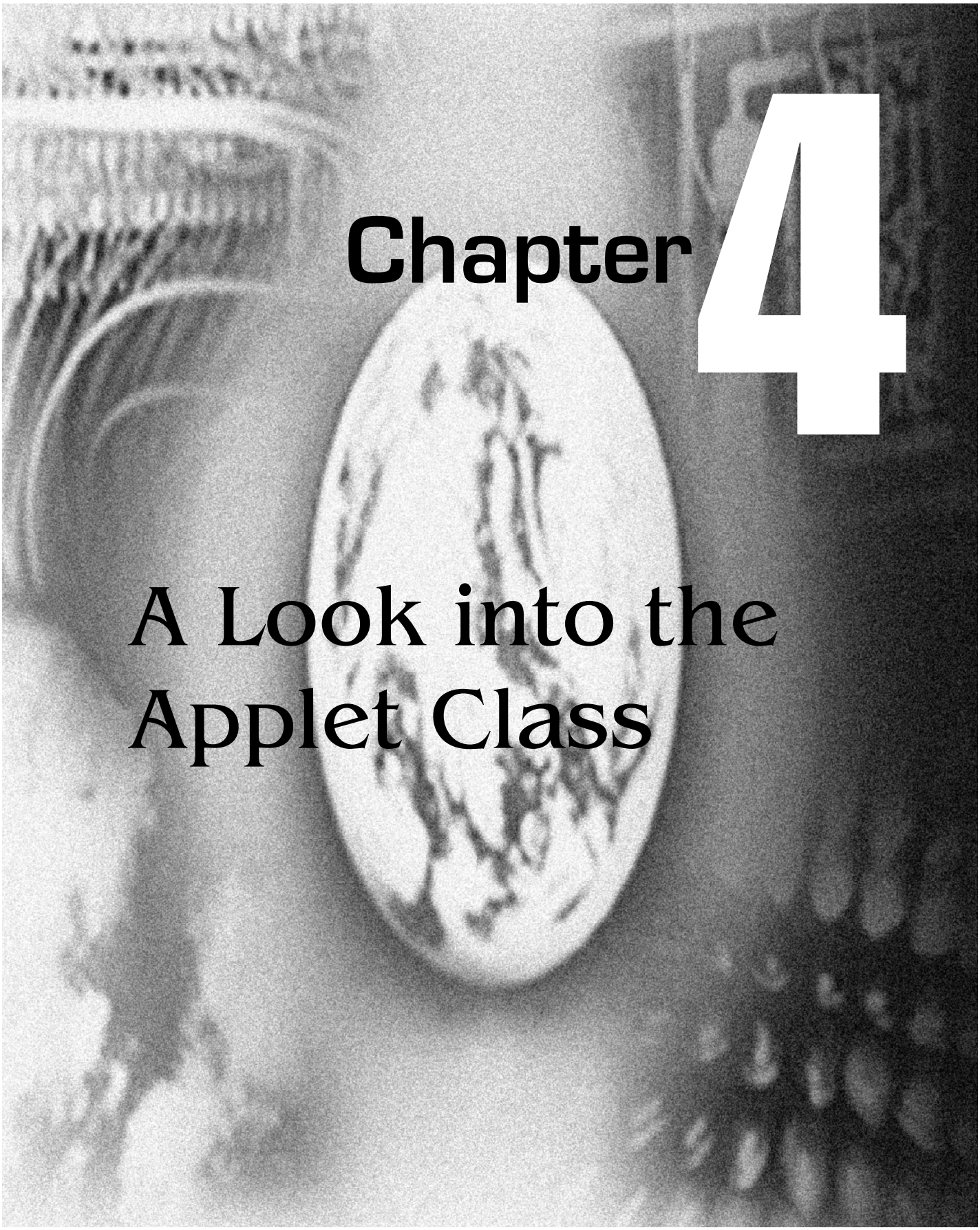
CD-ROM Included

WRITING JAVA APPLETS

- Master the art of creating distributed Internet programs
- Create your own powerful agent-server applications
- Use the Java source code and programming tools to create portable Java applets

John Rodley

CORIOLIS GROUP BOOKS



Chapter 4

A Look into the Applet Class

A Look into the Applet Class

4

The Applet class is your applets doorway into the browser. Check here as we cross the threshold.

Back in Chapter 1, we wrote a simple applet. In Chapter 3, we designed a system that presents itself on the Web by using an applet, the AgentLauncher, as its user interface. We have an idea of what an applet is, and a reason to use it. Now, it's time to get into more of the details.

In this chapter we'll write some applets and talk about where they appear, what they look like, when they run, and how they interact with their environment. When we're done, we should have a good idea of just what happens when a browser plows into that **<applet>** tag in an HTML document.

Positioning Applets on the Screen

As we've seen, the only way to get an applet to appear on the browser screen is by embedding an **<applet>** tag in an HTML document and then viewing that document. Like every other HTML element, the placement of the **<applet>** tag determines where on the screen the applet appears. Consider the simple example in Listing 4.1. and Listing 4.2, which provides the HTML code to run the applet, contains a number of HTML elements, including text paragraphs and images, our simple applet, then more HTML elements. Figure 4.1 shows how this applet appears running in Netscape.

Listing 4.1 Positioning an Applet

```
package chap4;

import java.awt.Graphics;
import java.applet.Applet;

/** An applet that prints the string ''Hello world'' at absolute
applet-relative coordinates x=20, y=50.
@author John Rodley
@version 1.0
*/
public class ch4_fig1 extends Applet {

/** Actually paint the string on the screen
*/
  public void paint(Graphics g) {
     g.drawString( ''Hello world'', 20, 50 );
  }
}
```

Listing 4.2 HTML Page Referencing Code

```
<!DOCTYPE HTML PUBLIC
''-//SQ//DTD HTML 2.0 HoTMetaL + extensions//EN''>
<HTML><HEAD><TITLE>An Applet in the HTML Milieu</TITLE></HEAD>
<BODY><H1>Some Text</H1>
<P>When you want to place an applet, you simply place it
within the page at the spot you want it to appear.  The
problem of placing applets is the same problem all HTML
elements have - the fact that HTML tags are 'suggestions', not
directives, which browsers are free to interpret
differently.  Now, a pair of goodogs, for your amusement:
<IMG SRC=''chap4/ch4_fig1a.jpg'' ALIGN=''BOTTOM''>
<IMG SRC=''chap4/ch4_fig1b.jpg'' ALIGN=''BOTTOM''>
<P>Below is the applet.  Can you tell the difference between
the applet text, and the HTML?</P>
<applet code=chap4/ch4_fig1.class width=100 height=100>
</applet>
The applet is above us.  Notice that as long as you don't
use non-default backgrounds, it's impossible to tell the
applet from the HTML.</P></BODY></HTML>
```

The two **import** statements used in Listing 4.1 are for the **Applet** class, which all applets subclass, and for the **Graphics** class, which we use in the **paint** method. The code line **extends Applet** tells the compiler that this class should get all the

functionality of the **Applet** class. The **paint** method overrides a do-nothing method that is part of the **Applet** class (via its inheritance of **Component**). The line **g.drawString** tells the **Graphics** object to draw a **String** at 20 on the X-axis and 50 on the Y. If you think about it, six lines of code (three of which are boilerplate stuff) have gotten us a functional and useful applet. We'll talk more about the **paint** method later in this chapter, and we'll cover the details of dealing with **Graphics** objects in more detail in Chapter 5. For now, what we're mainly interested in is how the applet fits into the host HTML document.

The rule browsers use when formatting a document is to stretch or compress the document vertically such that a line of text never rolls off the horizontal edge of the screen. If you resize the screen, Netscape will reformat the text *and* reposition the applet. As we resize the document in Figure 4.1 to make it narrower (smaller horizontally), our applet will eventually be pushed out the bottom of the window and we'll have to scroll vertically to get it back on the screen.

If you've played with browsers much, you've probably seen that browsers treat text and images differently. For instance, if you shrink the document in Figure 4.1 to make the window smaller horizontally than the width of our images,

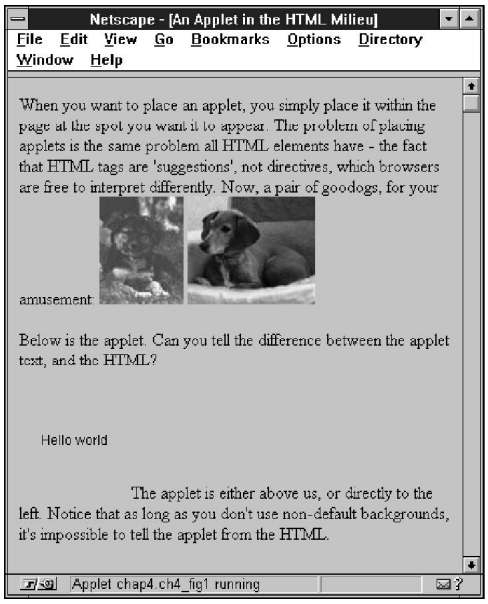

Netscape - [An Applet in the HTML Milieu]
File Edit View Go Bookmarks Options Directory
Window Help
When you want to place an applet, you simply place it within the page at the spot you want it to appear. The problem of placing applets is the same problem all HTML elements have - the fact that HTML tags are 'suggestions', not directives, which browsers are free to interpret differently. Now, a pair of goodogs, for your amusement:
Below is the applet. Can you tell the difference between the applet text, and the HTML?
Hello world
The applet is either above us, or directly to the left. Notice that as long as you don't use non-default backgrounds, it's impossible to tell the applet from the HTML.
Applet chap4.ch4_fig1 running

Figure 4.1
Netscape running Listing 4.2.

you'll see the browser add a horizontal scrollbar for scrolling the entire image. Browsers treat applets the same as images—as lumps that always get a rectangular area whose shape can't be changed by the browser.

Breaking down the Applet Class

So far, we've talked about applets in general terms—as small applications. But in Java, **Applet** is an actual class of its own, with very specific capabilities. In fact, the **Applet** class is at the end of a fairly long chain of inheritance consisting of the **Object**, **Component**, **Container**, **Panel**, and **Applet**. In simplistic terms, the first four superclasses give **Applet** the following capabilities:

- **Object** provides the ability to behave as a Java entity.
- **Component** provides the ability to appear as a visual entity on-screen.
- **Container** provides the ability to *encapsulate* other visual entities.
- **Panel** provides the ability to align encapsulated visual entities.

We've already talked a little about the basic Java **Object**. The remaining three classes are part of the windowing system, AWT, which we'll discuss briefly here, and will look at in more detail in Chapter 5. From these superclasses, we can easily infer a lot about what an applet is—it's a space on the screen, a window if you wish, that is a child of the window in which the browser is displaying the HTML text.

Component is the key class in this inheritance chain. Subclassing **Component** makes **Applet** a child window. Visually, a child window is a space on the screen that has the same background color as its parent, the browser main window. It has no border, and no menu, and nothing appears in that window unless we put it there. It can be drawn on at the pixel level (via **Graphics**) and it has the ability to capture events, like mouse clicks and key presses.

However, the **Applet** class is not just the sum of its superclasses. It implements many methods of its own, as shown in Table 4.1.

Over the course of this chapter, we'll use each of these methods, except for **getImage** and the URL grabbing methods, **getDocumentBase** and **getCodeBase**. If you're psyched to start displaying images via **getImage**, you'll have to wait until Chapter 5. We use both **getDocumentBase** and **getCodeBase** extensively in Chapter 7.

Table 4.1 The Applet Class

Method	Argument	Description
getApplet	String	Returns the applet or null if it isn't loaded
getAppletContext		Returns an object implementing the AppletContext interface
getAppletInfo		Returns a String that includes whatever information the applet author chooses; usually author, version, and copyright
getAudioClip	URL or URL,String	Returns an in-memory audio clip that can be played via Applet.play
getCodeBase		Returns the complete URL of this applet
getDocumentBase		Returns the complete URL of the HTML document in which this applet is embedded
getImage	URL or URL, String	Returns an in-memory image that can be drawn via Graphics.drawImage; this call *does not* load the image file over the Net
getParameter	String	Returns the string provided in the value=1portion of the param statement; for example, the argument <param name=GHIJ value=1234> returns returns "1234" after a call to getParameter("GHIJ")
getParameterInfo		Returns a two-dimensional String array explaining the arguments this applet accepts
init		Overload method that gets called whenever the applet is loaded
isActive		Returns true if Applet.start has been called
play	URL or URL,String	
resize	int height, width	
showStatus	String	
start		Overloaded method called each time the user visits the applet's HTML page
stop		Overloaded method called each time the user leaves the applet's HTML page

Resizing an Applet

The size of an applet is set through the **<applet>** tag in the HTML document. In Listing 4.2, we specified a height of 100 pixels and a width of 100 pixels, so the browser made a hole of that size in the document and flowed the text around it.

Unfortunately, some of the current crop of browsers don't deal well with reformatting the HTML document if the applet changes size. Suppose you use **Applet.resize** to make your applet smaller than the size set from the **<applet>** tag with **height** and **width** parameters. The HTML document should reflow to fit exactly around the new, smaller size. If the browser doesn't reflow the HTML document, there'll be an embarassing strip of default background in the spaces around the applet. Listing 4.3 changes our applet to downsize itself, uses a background color other than the default for the applet, and adds a background image to the HTML document. Figure 4.2 shows the improved applet running in Netscape.

Listing 4.3 A "Shriveled" Applet

```
package chap4;

import java.awt.Graphics;
import java.applet.Applet;
import java.awt.Color;

/** An applet that prints the string ''Hello world, again'' at
absolute applet-relative coordinates x=20, y=50 after clearing
the applet workspace to yellow.
@author John Rodley
@version 1.0
*/
public class ch4_fig2 extends Applet {

/** Resize the applet to slightly smaller than the applet
tag sizes us.
*/
  public void init() {
    resize( 90, 90 );
  }

/** Actually paint the string on the screen, first clearing the
workspace to yellow.
*/
  public void paint(Graphics g) {
```

```
    g.setColor(Color.yellow);
    g.fillRect(0,0,size().height, size().width);
    g.setColor(Color.black);
     g.drawString( ''Hello world, again'', 20, 50 );
  }
}
```

Listing 4.4 HTML Code That Runs the Shriveled Applet

```
<!DOCTYPE HTML PUBLIC
''-//SQ//DTD HTML 2.0 HoTMetaL + extensions//EN''>
<HTML><HEAD><TITLE>The Shrivelling Applet</TITLE></HEAD>
<BODY BACKGROUND=''background.jpg''>
<P>A pair of goodogs, to watch your incredible shrinking
applet:
<IMG SRC=''chap4/ch4_fig1a.jpg'' ALIGN=''BOTTOM''>
<IMG SRC=''chap4/ch4_fig1b.jpg'' ALIGN=''BOTTOM''>
Below is the applet, started with an initial size of 100 by
100 pixels:
<applet code=chap4/ch4_fig2.class width=100 height=100>
</applet>
The first word of this sentence should butt right up against
the applets right hand edge even though the applet has shrunk
to 90 by 90 pixels, assuming the browser re-flowed correctly.
</P></BODY></HTML>
```

Netscape - [The Shrivelling Applet]
File Edit View Go Bookmarks Options
Directory Window Help
A pair of goodogs, to watch your incredible
shrinking applet:
Below is the applet, started with an initial size of 100
Hello world, ag
by 100 pixels:
The first word of this sentence should butt right up against the applets right hand edge even though the applet has shrunk to 90 by 90 pixels, assuming the browser re-flowed correctly.
Document: Done

Figure 4.2
The shriveled applet running in Netscape.

So that's what happens if you shrink an applet within its allotted browser space. But what if you want to grow the applet beyond its browser space? It's the same issue in reverse: The HTML document has to be reflowed to provide more space for the growing applet. Listing 4.5 changes the applet in Listing 4.3 to burst out of its allotted space, and Figure 4.3 shows the results.

Listing 4.5 Growing an Applet

```
package chap4;

import java.awt.Graphics;
import java.applet.Applet;
import java.awt.Color;

/** An applet that prints the string ''Hello world, again'' at
absolute applet-relative coordinates x=20, y=50 on a red
background.
@author John Rodley
@version 1.0
*/
public class ch4_fig3 extends Applet {

/** Resize the applet to slightly larger than the applet
tag sizes us.
*/
  public void init() {
    resize( 120, 120 );
  }

/** Actually paint the string on the screen, after clearing the
applet to red first.
*/
  public void paint(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(0,0,size().height, size().width);
    g.setColor(Color.black);
     g.drawString( ''Hello world, again'', 20, 50 );
  }
}
```

Listing 4.6 HTML Code That Runs the Growing Applet

```
<!DOCTYPE HTML PUBLIC
''-//SQ//DTD HTML 2.0 HoTMetaL + extensions//EN''>
<HTML><HEAD><TITLE>The Growing Applet</TITLE></HEAD>
<BODY BACKGROUND=''background.jpg''>
```

```
<P>A pair of goodogs, to watch over your growing applet:
<IMG SRC=''chap4/ch4_fig1a.jpg'' ALIGN=''BOTTOM''>
<IMG SRC=''chap4/ch4_fig1b.jpg'' ALIGN=''BOTTOM''>
Below is the applet, started with an initial size of 100 by
100 pixels:
<applet code=chap4/ch4_fig3.class width=100 height=100>
</applet>
The first word of this sentence should butt right up against
the applets right hand edge even though the applet has grown to
120 by 120 pixels, assuming the browser re-flowed correctly.
</P></BODY></HTML>
```

As you can see, Netscape correctly reflows the document to accommodate whatever applet resizing you might do. Notice also that our text string doesn't completely print out. This is the fate of graphics operations that try to exceed the bounds of the applet: They simply don't happen. In Chapter 5's ticker example, we'll take a closer look at how to fit text into a given area.

When Does an Applet Run?

For traditional computer programs the answer to the question "when does the program run?" is obvious. When the user runs the program, it runs, and when the user closes it, it stops running. The user doesn't have that kind of control over an applet. It is the browser, not the user, that loads, runs, stops, and unloads applets, so the answer becomes a lot more involved. The simplistic answer to the

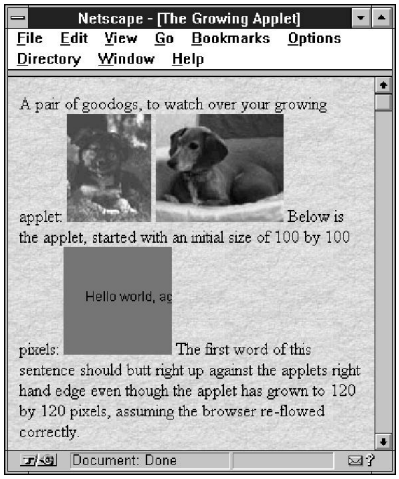

Netscape - [The Growing Applet]
File Edit View Go Bookmarks Options
Directory Window Help
A pair of goodogs, to watch over your growing
applet:
Below is
the applet, started with an initial size of 100 by 100
Hello world, ag
pixels:
The first word of this
sentence should butt right up against the applets right
hand edge even though the applet has grown to 120
by 120 pixels, assuming the browser re-flowed
correctly.
Document: Done

Figure 4.3
The growing applet running in Netscape.

question is this: An applet runs when its space in the HTML document is visible, and stops running when, for whatever reason, that space in the HTML document is not visible.

Unless an applet creates and runs in its own thread, the simple answer holds true. This is because an applet like the simple one shown in Listing 4.1 is running in a thread that the browser has created and controls. When Java applets create their own threads, the answer becomes more complicated. Technically, because these threads are beyond the control of the browser, there is no written-in-stone rule on when an applet runs. In theory, browsers expect well-behaved applets to adhere to the basic rule of run when you're on-screen, and stop running when you're off-screen, but whether or not an applet implements that theory is completely up to the applet writer. We'll discuss threads in more detail in Chapter 6, and eventually write our own multi-threaded and ill-behaved, but wickedly useful, applets.

Whether the applet itself is single or multi-thread, ill or well-behaved, the browser obeys a certain set of rules when trying to run it. In this sense, the browser treats the applet as a peer, where, no matter how the applet behaves, it can expect certain behaviors from the browser. These behaviors, the browser invoking one of four applet methods, are clues as to what is going on in the browser.

Here's a brief rundown of the methods: When the browser first encounters the **<applet>** tag in the HTML doc, it calls **Applet.init**. When a user visits the applet's page, the browser calls **Applet.start**. When the user leaves the applet's page, the browser calls **Applet.stop**, and whenever it decides the applet is not needed anymore and should be expunged from memory, it calls **Applet.destroy**. Thus, **Applet.init** and **Applet.destroy** get called once in the life of an applet, while **Applet.start** and **Applet.stop** can be called any number of times. An applet that cares about whether or not it's loaded or on-screen can override **init**, **start**, **stop** and **destroy**, as shown in Figure 4.4.

Adding Sound to an Applet

Let's improve our simple applet to sing us a little song. Listing 4.7 shows these modifications. Play with the applet. See what happens if you scroll the applet off the screen, or if you hyperlink to another page.

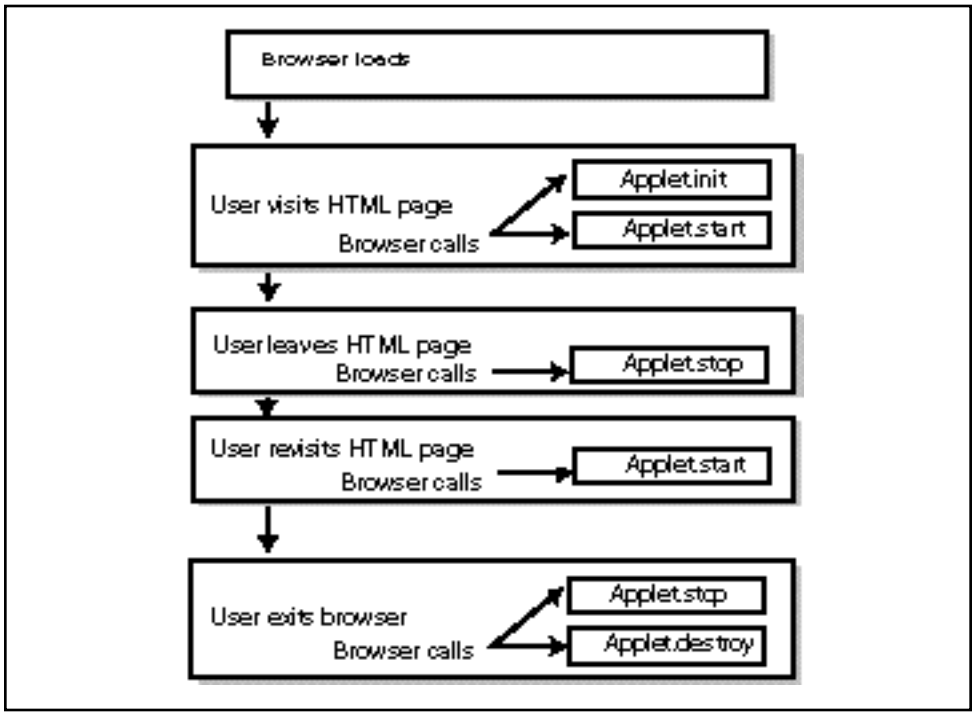


Figure 4.4
Overridden methods along the applet execution trail.

Listing 4.7 The Do-Nothing Applet, with Sound

```
package chap4;

import java.awt.Graphics;
import java.applet.Applet;
import java.applet.AudioClip;
import java.net.*;

/** A simple app that plays a tune, supplied by the applet tag
via the name ''StartClip'' every time Applet.start is called, and
plays another tune, paramater name ''EndClip'' every time
Applet.stop is called. Plays the RunningClip in a loop between
start and stop.
@version 1.0 12/20/1995
@author John Rodley
*/
public class ch4_fig4 extends Applet {
```

```
  String sAppletInfo;
  URL StartClipURL;
  URL EndClipURL;
  AudioClip RunningClip;

  String pinfo[][] = {
     {''StartClip'', ''URL'', ''The URL of an audio clip played at
startup'' },
     {''RunningClip'', ''URL'',
        ''The URL of an audio clip played between start and stop'' },
     {''EndClip'', ''URL'', ''The URL of an audio clip played at stop'' }
      };

/** Whenever a user visits our page, the browser calls this.
This is an override of an Applet method.. */
  public void start() {
    if( RunningClip == null || StartClipURL == null )
      return;
    play( StartClipURL );
    RunningClip.loop();
    System.out.println( ''after loop'' );
  }

/** Whenever the user leaves our page, this gets called.
Another override of an Applet method.
*/
  public void stop() {
    if( RunningClip == null || EndClipURL == null )
      return;
    RunningClip.stop();
   play( EndClipURL );
  }

/** Tell anyone who might inquire what we are.
@return A string describing this applet.
*/
  public String getAppletInfo() {
   return(''JRodley - Java Applets Book - Chapter 4 figure 4'');
     }

/** Tell anyone who might inquire how to use this applet.
@return A 2d string array describing the arguments to this
applet.
*/
  public String[][] getParameterInfo() {
     return pinfo;
   }
```

```
/** Connect to the URLs specified by the StartClip and EndClip
parameters in the applet tag.
*/
  public void init() {
   try {
      StartClipURL = new URL(getParameter(''StartClip''));
      RunningClip = getAudioClip(new URL(getParameter(''RunningClip'')));
      EndClipURL = new URL(getParameter(''EndClip''));
      } catch( MalformedURLException e )
           { System.out.println(''exception''+e );}
  }
}
```

Listing 4.8 HTML That Runs the Do-Nothing Applet

```
<!DOCTYPE HTML PUBLIC
''-//SQ//DTD HTML 2.0 HoTMetaL + extensions//EN''>
<HTML><HEAD><TITLE>The Singing Applet</TITLE></HEAD>
<BODY>
<applet code=chap4/ch4_fig4.class width=100 height=100>
<param name=StartClip value=''http://www.coriolis.com/japp/chap4/
charge.au''>
<param name=RunningClip value=''http://www.coriolis.com/japp/chap4/
charge.au''>
<param name=EndClip value=''http://www.coriolis.com/japp/chap4/
retreat.au''>
</applet>
</P></BODY></HTML>
```

Audio and Parameters

The basis of Java's audio capability is the **AudioClip** class. In order to do any audio, you need to get the actual audio data from a file somewhere on the Net. Where is that file? We could hardwire the file name into the class file, as in the following code snippet:

```
public final static string startClipFile =
new String( ''http://www.channel1.com/users/ajrodley/start.au'' );
```

However, a more flexible way of dealing with data like this is to pass it in as an argument via the **<param name=value=>** construct within the HTML source. In Listing 4.8, we've defined three arguments—**StartClip**, **RunningClip**, and **EndClip**—each of which is the URL of an audio file somewhere out on the Net. In the **init** method, which as we've said is the first override method invoked by

the browser, we query these arguments via **Applet.getParameter**. When, in our HTML page, we say:

```
<param name=StartClip value=''http://www.channel1.com/users/ajrodley/
start.au''>
```

the call:

```
String s = getParameter(''StartClip'');
```

returns the **String** http://www.channel1.com/users/ajrodley/start.au.

Applets, as we'll discuss later, operate in a multi-applet environment. This is a good thing, but it imposes some responsibilities on well-behaved applets. The first of these is the responsibility to document the arguments the applet accepts via the **getParameterInfo** method. Listing 4.4 implements both this and the **getAppletInfo** method, which allows users of our applet to find information about the applet's author, its version, and whatever else we want to put in the returned string. These methods are not required, but are courtesies we extend to other applets that may want to know something about us, or to HTML and applet writers who may want to use our applet in their own code.

The singing applet uses two different methods for playing audio clips. The start and end clips only need to play one time through, whenever their method (**start** or **stop**) is invoked, so we can play them simply by passing their URLs to **Applet.play**. We want the running clip to play continuously, though. For that, we need explicitly to create an **AudioClip** object. Thus, in our **init** method, we create three objects: a **URL** for the start clip, a **URL** for the end clip, and an **AudioClip** for the running clip.

The **init** method also contains a new wrinkle for us, an exception handling clause. We deal with exceptions in more detail in Chapter 8. What you need to remember here is that if a method is defined as "throwing" an exception, any method that calls it has to either define itself as *also* throwing that exception, or it has to handle that exception with a **try-catch** statement. Here, we use the latter method. The **URL** constructor throws **MalformedURLException**, so we catch it, and simply make a note of it on standard output. The **URL** constructor throws an exception if the specified URL is not accessible. Thus, if our **init** method runs without throwing an exception, we can be sure that there are accessible files at the URLs we specified.

When **init** is done, we have three objects: two **URL**s and one **AudioClip**. The **URL** constructors only made sure there was a file at the URL but the **AudioClip** constructor actually fetched the data from the URL. To play the start and end clips within the **start** and **stop** methods, we simply invoke the **play** method with the proper audio file's **URL** as the argument. To keep the running clip playing continuously while the applet is on-screen, we simply invoke **AudioClip.loop** in the **start** method. **AudioClip.loop** returns immediately, even though the clip continues to play until **AudioClip.stop** in our applet's **stop** method. Listing 4.9 presents a slightly different version of our do-nothing applet.

Listing 4.9 The Do-Nothing Applet in Its Own Thread

```
package chap4;

import java.util.*;
import java.applet.*;
import java.awt.*;

/* A do-nothing applet that runs in its own thread.

@version x.xx 1.10, 1 August, 1995 xxx
@author John Rodley
*/

public class ch4_fig5 extends Applet implements Runnable {
/** The main thread. */
Thread myThread;
String TheString;
int iteration = 0;

/** Start the main thread for this game. */
  public void start() {
    TheString = new String( ''Applet is on screen now in '' );
    if( myThread == null )
       {
    myThread = new Thread( this );
    myThread.start();
      }
  }

/** Stop this thread. */
  public void stop() {
     TheString =
      new String(''Applet is off screen but still running in '');
  }
```

```
/** The main loop for the main thread. */
  public void run() {
     while( myThread != null )
        {
        try {
        Thread.sleep( 300 );
          } catch( InterruptedException e )
          { System.out.println(''exception'' ); }
        showStatus( TheString+iteration+''th iteration '');
        iteration++;
        }
     myThread = null;
   }
}
```

In this example, we run the applet in its own thread. The key things to note here are the **start** and **run** methods, and the fact that our applet implements the **Runnable** interface. In the **start** method, we push the applet into its own thread by creating a new **Thread** object and passing the applet itself as the only argument to the constructor. This is possible only because our applet implements **Runnable**.

The only requirement of the **Runnable** interface is that you implement a **run** method, as we have here. Sometime after the call to **myThread.start** returns, the **run** method will begin executing asynchronously. Within this **run** method, we simply sit in a loop, sleeping for 300 milliseconds, then incrementing a counter and displaying it in the browser status line. Note that because we only want *one* of these independent threads to run, we check **myThread** within **start**. Otherwise, every visit to this page would create a new thread with the predictable effect on system performance.

Run this applet under Netscape. You'll see the status string changing regularly. Now, hyperlink away to some other HTML page. Notice that the status string keeps changing. We have violated one of the rules of well-behaved applets. Our host HTML document has disappeared, but our applet is still running!

Our new do-nothing applet will continue to run for as long as the browser is up. To kill it, you have to close Netscape. This persistence presents ominous possibilities. In Listing 4.10, a simple change turns the do-nothing applet into a malicious do-nothing applet.

Note: Do not run the applet shown in Listing 4.10 or you will have to kill your browser to get rid of it.

Listing 4.10 Eating CPU Cycles

```
package chap4;

import java.util.*;
import java.applet.*;
import java.awt.*;

/* A malicious do-nothing applet that runs in its own thread,
eating up CPU cycles because it neither sleeps nor exits.

@version x.xx 1.10, 1 August, 1995 xxx
@author John Rodley
*/

public class ch4_fig6 extends Applet implements Runnable {

/** The main thread. */
Thread myThread;
String TheString;
int iteration = 0;

/** Start the main thread for this game. */
  public void start() {
    TheString = new String( ''Applet is on screen now in '' );
    if( myThread == null )
       {
     myThread = new Thread( this );
     myThread.start();
       }
  }

/** Stop this thread. */
  public void stop() {
     TheString =
      new String(''Applet is off screen but still running in '');
  }

/** The main loop for the main thread. */
  public void run() {
     while( myThread != null )
       {
       showStatus( TheString+iteration+''th iteration '');
```

```
        iteration++;
        }
      myThread = null;
   }
}
```

The malicious applet eats up a lot of CPU cycles when it is run. Let's take a close look at the code changes in this listing to see why. We removed the **sleep** call from the **run** method, which results in us simply sit in a loop incrementing a variable. We also removed the call to **showStatus,** which alerts the user that we are still running. If you were to run this applet in Netscape and then hyperlink to another page, you would notice how unresponsive the browser becomes. The key features of this malicious applet are actually omissions, the fact that our **run** method never blocks, and that we didn't implement an **Applet.stop** method that stops our thread. All applets that run in their own thread must take special care to run only when, and only as much as neccessary.

By this point, you've seen the light and want to write well-behaved, threaded applets. But how? Listing 4.11 modifies the threaded do-nothing applet to stop running when the user leaves the page, and then re-start when the user returns.

Listing 4.11 A Well-Behaved Do-Nothing Applet

```
package chap4;

import java.util.*;
import java.applet.*;
import java.awt.*;

/* A do-nothing applet that runs in its own thread.

@version x.xx 1.10, 1 August, 1995 xxx
@author John Rodley
*/

public class ch4_fig7 extends Applet implements Runnable {

/** The main thread. */
Thread myThread;
String TheString;
int iteration = 0;
```

```
/** Start the main thread for this game. */
  public void start() {
    TheString = new String( ''Applet is on screen now in '' );
    if( myThread == null )
       {
    myThread = new Thread( this );
    myThread.start();
       }
   }

/** Stop this thread. */
  public void stop() {
     TheString =
      new String(''Applet is off screen but still running in '');
// This line is the key. If myThread is null, the run method
// returns and the thread disappears.
    myThread = null;
  }

/** The main loop for the main thread. */
  public void run() {
     while( myThread != null )
       {
       try {
       Thread.sleep( 300 );
          } catch( InterruptedException e )
          { System.out.println(''exception'' ); }
       showStatus( TheString+iteration+''th iteration '');
       iteration++;
       }
   }
}
```

With a threaded applet, the thread the applet runs in only exists for as long as the **run** method is running. What we need is code in our **stop** method (which the browser calls whenever the user leaves the page) that will cause the **run** method to break out of its loop. This is what setting **myThread** to **null** does. Then in the **start** method (called whenever the user visits our page) we re-create the thread and set it running. We'll talk more about threads in Chapter 6. In the meantime, the well-behaved, threaded do-nothing applet will serve as a good template for any threaded applets we might need.

Making Our Applets Interact with the Browser

Prior to the beta version of Java, applets existed in a vacuum. They had no way of knowing anything about any other applets. Now, applets can call **Applet.getAppletContext**. This call returns an object that implements the **AppletContext** interface. Usually, this object will be the browser itself, but it might not be. In fact, browsers that wish to protect themselves from rogue applets may create a separate class just to contain the applet context for this call. In any case, it doesn't matter to us because, as with any interface, the definition of the interface is all that you can rely on.

The **AppletContext** class, shown in Table 4.2, provides six methods, which allow an applet to get images and audio clips, learn about other applets active in the browser, and show new HTML documents.

Using **getAppletContext**, we can modify our well-behaved, threaded do-nothing applet to get information about all the other running applets. Listing 4.12 shows our new applet, while Listing 4.13 shows the HTML code used to create four of these applets on one Web page. Figure 4.5 shows the output of our new applet.

Table 4.2 The AppletContext Interface

Method	Argument	Description	Return
getApplet	String	Returns the applet, or null if It isn't loaded	Applet
getApplets		Returns a list of the applets loaded and casts the members of the Enumeration to Applet to use them	Enumeration
getAudioClip	URL	Returns a playable audio stream	AudioClip
getImage	URL	Returns an in-memory image that can be drawn via Graphics.drawImage	Image
showDocument	URL		
showStatus	String	Returns the message to be displayed in the status bar along the bottom of the browser window	

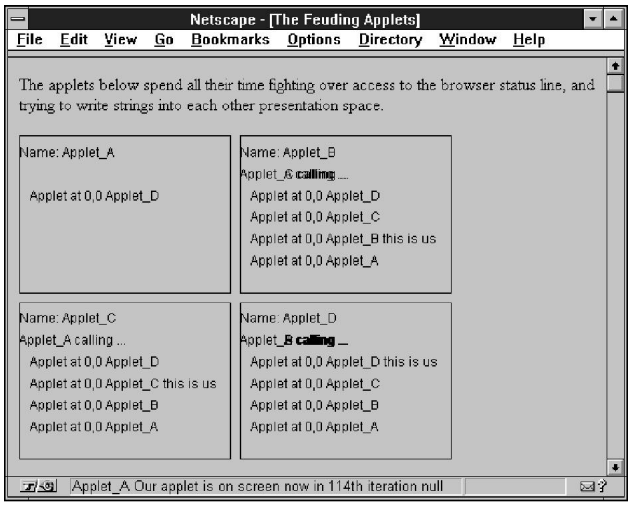

Netscape - [The Feuding Applets]
File Edit View Go Bookmarks Options Directory Window Help
The applets below spend all their time fighting over access to the browser status line, and trying to write strings into each other presentation space.
Name: Applet_A
Applet at 0,0 Applet_D
Name: Applet_B
Applet_A calling ...
Applet at 0,0 Applet_D
Applet at 0,0 Applet_C
Applet at 0,0 Applet_B this is us
Applet at 0,0 Applet_A
Name: Applet_C
Applet_A calling ...
Applet at 0,0 Applet_D
Applet at 0,0 Applet_C this is us
Applet at 0,0 Applet_B
Applet at 0,0 Applet_A
Name: Applet_D
Applet_B calling ...
Applet at 0,0 Applet_D this is us
Applet at 0,0 Applet_C
Applet at 0,0 Applet_B
Applet at 0,0 Applet_A
Applet_A Our applet is on screen now in 114th iteration null

Figure 4.5
The output of the new simple applet.

Listing 4.12 Finding Other Applets

```
package chap4;

import java.util.*;
import java.io.*;
import java.net.*;
import java.applet.*;
import java.awt.*;
/** An applet that finds the other applets that exist within
this AppletContext, fights with them to write a string to the
browser status line, and writes a string directly into the
other applet windows.

@version 1.10, 11/20/95
@author John Rodley
*/

public class ch4_fig8 extends Applet implements Runnable {


/** The main thread. */
Thread myThread = null;
String TheString = null;
```

```
String myName;
String paintString = null;
int iteration = 0;
int paintIteration = 0;

/** Start the main thread for this game. */
  public void start() {
     TheString = new String( ''Our applet is on screen now in '' );
   if( myThread == null )
      {
        myThread = new Thread( this );
      myThread.start();
      }
   }
/** Stop this thread. */
  public void stop() {
     myThread = null;
   }

/** The main loop for the main thread. */
  public void run() {
     while( myThread != null )
        {
        try {
        Thread.sleep( 300 );
           } catch( InterruptedException e )
              { System.out.println(''exception'' ); }
        showStatus( myName+'' ''+TheString+iteration+
                       ''th iteration ''+paintString );
        repaint();
        iteration++;
        }
   }

/** Paint our window.  Display a string in this window for
each applet we find, then display a string in each of the other
applet's windows.
*/
public void paint( Graphics g ) {
   String gString;
   int i = 0;

   // Label this applet with its name
   g.drawString( ''Name: ''+myName, 0, 20 );

   AppletContext ac = getAppletContext();
```

```
   Enumeration e = ac.getApplets();
   Dimension d = size();
   g.drawRect( 0, 0, d.width-5, d.height-5 );
   while( e.hasMoreElements()) {
      Applet a = (Applet)(e.nextElement());
      // If this applet is a ch4_fig8, then it has a name parameter set
      String s = a.getParameter(''Name'');
      if( s == null )
         s = new String( ''not a ch4_fig8 applet'' );
      Point p = a.location();
      gString = new String( ''Applet at ''+p.x+'',''+p.y+'' ''+s );
      // The list will always include ''this'', so check it
      if( a == this )
         gString = new String( gString+'' this is us'' );
      else {
         // Now draw a string into the other applets window
         Graphics g1 = a.getGraphics();
         g1.drawString( myName+'' calling ...'', 0, 40 );
         }
      // Draw the string describing the other applet in our window
      g.drawString( gString, 10, 20*(i+3) );

      i++;
      }
   }

/** If the user clicks on the applet, stop it.
@return false
*/
  public boolean mouseDown(Event e, int x, int y) {
   stop();
     return( false );
  }

/** Initialize the applet. Resize it and give it a name.  Make
sure that the applet is ''named'' via the applet tag.
*/
  public void init() {
   myName = getParameter( ''Name'' );
     if( myName == null )
        myName = new String( ''Set the Name= parameter please!'' );
   resize( 210, 210 );
   }
}
```

The **AppletContext** interface provides the method **getApplets**, which returns an **Enumeration** (basically a vector) containing all the applets running in this **AppletContext**. We retrieve each individual applet via **Enumeration.nextElement**.

The applet returned by **getApplets** is completely capable. We can call any method available in the **Applet** class against that applet. Thus, in our **paint** method, we call **drawString** against our own applet, and against each of the other applets we find. As this page runs, you can see the four applets dueling over the status line and drawing over each other as they draw the intrusive string into each other's graphics context.

In Listing 4.12, we introduce a new class, **Point** and a new interface, **Enumeration**. **Point** simply encapsulates two **int**s, the X and Y coordinates of the point. In C or C++, this would undoubtedly have been implemented as a simple **struct**. **Enumeration** is a far more interesting concept. What does it mean to enumerate something? Here, enumeration means "to walk through a list, using up the elements as you go." Essentially, an **Enumeration** is a list that you can go through once and only once. The interface requires only two methods, **hasMoreElements** and **nextElement**, which force you to use the same flow-control whenever you're dealing with an **Enumeration**. The technique is illustrated in the following code snippet:

```
// assume e is an object that implements the Enumeration interface
while( e.hasMoreElements()) {
  Object o = e.nextElement();
  ...
  }
// At this point e has been ''used up.''  It cannot be walked through
again.
```

In Listing 4.12's **paint** method, we get an **Enumeration** of **Applets** from the **AppletContext** interface method **getApplets**, then walk through it, dueling with each of the applets it returns. The following code line illustrates one of the powers of Java's object orientation that might not be obvious:

```
Applet a = (Applet )(e.nextElement());
```

This power works like this: **Enumeration.nextElement** always returns an **Object**, but a simple cast of that **Object** to **Applet**, gives us access to the much wider

range of capabilities of the **Applet** class. That **Object** was always an **Applet**. It's just that the **Enumeration** doesn't *care* what class its elements are, so it stores them as **Object** and lets the user of the **Enumeration** deal with casting them back to the proper type. What would happen if, by some chance, the call to **e.nextElement** returned an **Object** that was *not* of the **Applet** class? The Java interpreter would throw an **IllegalCastException**. This is something you have to consider very carefully whenever you cast from a base class to a superclass, especially when you're using **Vectors** or **Enumerations** you get from someone else's objects.

With that bit of type-casting wisdom in mind, consider the following code snippet from Listing 4.12:

```
Applet a = (Applet)(e.nextElement());
// If this applet is a ch4_fig8, then it has a name parameter set
String s = a.getParameter(''Name'');
if( s == null )
   s = new String( ''not a ch4_fig8 applet'');
```

This code simply determines whether we're dealing with the proper class of object, a **ch4_fig8**. However, there's a much better way to deal with this, as shown in the following code snippet:

```
Applet a = (Applet)(e.nextElement());
// If a is not a ch4_fig8, move on to the next applet
if( a instanceof ch4_fig8 )
   ...
else
   continue;
```

The **instanceof** operator tells us whether **Object a** is an instance of class **ch4_fig8**. Checking objects with the **instanceof** operator is usually a good idea *before* you try a type cast. In this particular instance, **AppletContext.getApplets** guarantees that it will return an **Enumeration** of **Applets**, so the cast to **Applet** should be safe, but beyond that, we can't be sure, so we use **instanceof**.

AppletContext.getApplets gives us an inter-applet communication mechanism for which there are any number of uses. As another brutally simple example, let's modify the dueling applet of Listing 4.12 so that it kills off all the other applets on a page via **Applet.stop**, ensuring that it's the only running applet on a page.

Listing 4.13 A "Killer" Applet

```
package chap4;

import java.util.*;
import java.applet.*;
import java.awt.*;

/** An applet that finds the other applets that exist within
this AppletContext, and stops them.

@version 1.10, 11/20/95
@author John Rodley
*/

public class ch4_fig9 extends Applet implements Runnable {

/** The main thread. */
Thread myThread = null;
String TheString = null;
int iteration = 0;
int paintIteration = 0;

/** Start the main thread for this game. */
  public void start() {
     TheString = new String( ''Our applet is on screen now in '' );
   if( myThread == null )
      {
        myThread = new Thread( this );
      myThread.start();
      }
  }

/** Resize us to be wide enough for the status string. */
  public void init() {
    resize( 300, 100 );
  }

/** Stop this thread. */
  public void stop() {
     myThread = null;
   }

/** The main loop for the main thread. */
  public void run() {
     while( myThread != null )
        {
```

```
       try {
       Thread.sleep( 300 );
          } catch( InterruptedException e )
            { System.out.println(''exception'' ); }
       showStatus( iteration+''th iteration '' );
       repaint();
       iteration++;
       }
  }
/** Paint our window.  Destroy each applet we find,
then display a string boasting of this achievement.
*/
  public void paint( Graphics g ) {
    String gString;
   int i = 0;

    AppletContext ac = getAppletContext();
   Enumeration e = ac.getApplets();
    while( e.hasMoreElements()) {
      Applet a = (Applet)(e.nextElement());
     Point p = a.location();
     // The list will always include ''this'', so check it
       if( a instanceof ch4_fig9 )
        gString = new String( ''this is us'' );
     else {
         Container c = a.getParent();
         if( c == null ) // this applet was already removed
           continue;
        // Now destroy the other applet
         a.stop();
         c.remove(a);
       gString = new String( ''Stopped applet at ''+p.x+'',''+p.y+''
''+iteration);
       }
      g.drawString( gString, 10, 20*(i+3) );

       i++;
   }

  }
}
```

Listing 4.14 HTML Code That Runs the Killer Applet

```
<!DOCTYPE HTML PUBLIC
''-//SQ//DTD HTML 2.0 HoTMetaL + extensions//EN''>
<HTML><HEAD><TITLE>The Killer Applet</TITLE>
```

```
</HEAD>
<BODY>
<P>
The applets below spend all their time fighting over access to
the browser status line, and trying to write strings into each
other's presentation space.
</P>
<applet code=chap4/ch4_fig8a.class width=100 height=100>
<param name=Name value=Applet_A>
</applet>
<applet code=chap4/ch4_fig8a.class width=200 height=100>
<param name=Name value=Applet_B>
</applet>
<applet code=chap4/ch4_fig8a.class width=300 height=100>
<param name=Name value=Applet_C>
</applet>
<applet code=chap4/ch4_fig9.class width=400 height=100>
</applet>
</BODY></HTML>
```

Try running this applet and you'll see that only the killer applet remains running. Notice also that, not only do the applets stop running, but their screen output disappears. While we'll talk more about removing **Components** in the next section, it's important to note that even though we've stopped the other applets via the **stop** method, those applets continue to exist, and the **AppletContext** still returns them every time we call **getApplets**. That's why we check the return from **getParent**. Calling **Container.remove** on a **Component** means that the next call to **getParent** will return a null. That's how we know that the **Component** is no longer displayed.

Controlling Stopped Applets

What to do with stopped applets is a more interesting question than it might seem. In Listing 4.12, we added a method, **mouseDown**, that stops the applet if the user clicks on it. This method is shown in the following code snippet:

```
public boolean mouseDown( Event e, int x, int y ) {
  stop();
  return( false );
}
```

Run Listing 4.12 once again and try clicking on one of the applets. What you see is that the applet does stop running, but that its last "state" remains on the

screen. In the **AppletContext**'s (browser's) view of the world, the stopped applet still exists, and the allocated section of the browser window should remain uncleared and unused. Try stopping one of the applets, then moving the browser window around. What you'll see is that the applet hasn't really gone away at all. All that's happened is that the **run** method has returned. When the applet window gets covered and then uncovered, Java decides that all the applets need to be repainted and it calls **Applet.paint** for all the applets on the screen. What we need to do is remove the applet from the list of things that get repainted. To do this, we add a single line to **Applet.stop**, as shown in Listing 4.15.

Listing 4.15 Clearing the Applet's Window

```
package chap4;

import java.util.*;
import java.io.*;
import java.net.*;
import java.applet.*;
import java.awt.*;

/** An applet that finds the other applets that exist within
this AppletContext, fights with them to write a string to the
browser status line, and writes a string directly into the
other applet windows.  Modified to stop and 'disappear' from
the screen if the user clicks on the applet.

@version 1.10, 11/20/95
@author John Rodley
*/

public class ch4_fig11 extends Applet implements Runnable {


/** The main thread. */
Thread myThread = null;
String TheString = null;
String myName;
String paintString = null;
int iteration = 0;
int paintIteration = 0;

/** Start the main thread for this game. */
  public void start() {
     TheString = new String( ''Our applet is on screen now in '' );
```

```
  if( myThread == null )
    {
      myThread = new Thread( this );
    myThread.start();
    }
  }


/** Stop this thread. */
  public void stop() {
    myThread = null;
  }

/** The main loop for the main thread. */
  public void run() {
     while( myThread != null )
       {
       try {
       Thread.sleep( 300 );
          } catch( InterruptedException e )
             { System.out.println(''exception'' ); }
       showStatus( myName+'' ''+TheString+iteration+
                     ''th iteration ''+paintString );
       repaint();
       iteration++;
       }
  }

/** Paint our window.  Display a string in this window for
each applet we find, then display a string in each of the other
applet's windows.
*/
public void paint( Graphics g ) {
  String gString;
  int i = 0;

  // Label this applet with its name
  g.drawString( ''Name: ''+myName, 0, 20 );

  AppletContext ac = getAppletContext();
  Enumeration e = ac.getApplets();
  Dimension d = size();
  g.drawRect( 0, 0, d.width-5, d.height-5 );
  while( e.hasMoreElements()) {
    Applet a = (Applet)(e.nextElement());
    // If this applet is a ch4_fig8, it has a name parameter set
    String s = a.getParameter(''Name'');
```

```
    if( s == null )
      s = new String( ''not a ch4_fig8 applet'' );
    Point p = a.location();
    gString = new String( ''Applet at ''+p.x+'',''+p.y+'' ''+s );
    // The list will always include ''this'', so check it
    if( a == this )
      gString = new String( gString+'' this is us'' );
    else {
      // Now draw a string into the other applet's window
      Graphics g1 = a.getGraphics();
     // If this applet's already stopped, g1 will be null
     if( g1 != null )
      g1.drawString( myName+'' calling ...'', 0, 40 );
      }
    // draw the string describing the other applet in our window
    g.drawString( gString, 10, 20*(i+3) );

    i++;
    }
  }

/** If the user clicks on the applet, stop it.
@return false
*/
  public boolean mouseDown(Event e, int x, int y) {
   stop();
    getParent().remove(this);
    return( false );
  }

/** Initialize the applet. Resize it and give it a name.  Make
sure that the applet is ''named'' via the applet tag.
*/
  public void init() {
   myName = getParameter( ''Name'' );
    if( myName == null )
      myName = new String( ''Set the Name= parameter please!'' );
   resize( 210, 210 );
   }
}
```

Our addition to **Applet.stop** clears the space that our **Applet** once occupied, and removes it from the list of **Components**, ensuring that it never gets repainted. Removing the applet **Component** from the **Container** it resides in results in an applet that has no screen representation. This means that *any* call to

that applet's **Component** superclass, such as the **paint** method's call to **a.getGraphics**, will probably fail. We take this into account in **paint** by adding a **null** return check to **a.getGraphics**.

In truth, you can't really get rid of an applet. There's no **delete** operator in Java, so you can't just destroy it, and the garbage collector won't get rid of it automatically until there are no more references to it. The applet was created (via the **new** operation) by the browser, and the applet itself doesn't have write access to the browser's list of applets, so there will always be that one reference to the applet. For now, the best we can do is to remove the applet **Component** from the browser's list of **Components**, and hope that the browser (whichever one it is) will reuse the now-available screen space. We'll talk more about **Components** and **Containers** in Chapter 5.

Applets and Hyperlinks

We can combine our persistent applet with the **AppletContext** to implement a rudimentary Web trolling applet. This applet takes a list of URLs and instructs the **AppletContext** to visit those URLs, in sequence, with a delay of 30 seconds at each document.

Listing 4.16 A Web-Crawling Applet

```
package chap4;

import java.util.*;
import java.io.*;
import java.net.*;
import java.applet.*;
import java.awt.*;

/* A threaded applet that takes a list of URLs as an argument,
then visits each of those URLs, delaying for delay milliseconds
between URL switches.

@version 1.0 12/23/95
@author John Rodley
 */

public class ch4_fig12 extends Applet implements Runnable {

/** The main thread. */
Thread myThread = null;
```

```
int iteration = 0;
Vector docURL;
int delay = 10000;

/** Start the main thread for this game. */
public void start() {
  if( myThread == null )
    {
    myThread = new Thread( this );
  myThread.start();
    }
  }


/** Stop this thread. */
public void stop() {
  myThread = null;
  }

/** The main loop for the main thread. */
public void run() {
  System.out.println( ''Runnable.run'' );
  iteration = 0;
    while( myThread != null )
      {
      AppletContext ac = getAppletContext();
      showStatus( ''Switching to ''+docURL.elementAt(iteration));
      ac.showDocument((URL) docURL.elementAt(iteration ));
      try {
      Thread.sleep( delay );
        } catch( InterruptedException e ) {
System.out.println(''exception'' ); }
      iteration++;
      if( iteration >= docURL.size() )
        iteration = 0;
      }
    myThread = null;
  }

/** Initialize the applet. Resize and load images.
*/
public void init() {
  docURL = new Vector(1);
  int i = 1;
  String sDelay = getParameter(''Delay'');
  if( sDelay != null )
    delay = new Integer( sDelay ).intValue();
```

```
  while( true ) {
    String s = getParameter(''Document''+i);
    if( s == null ) break;
    i++;
    try {
    docURL.addElement( new URL(s));
      } catch( Exception e) { break; }
    }
  resize( 210, 210 );
  }
}
```

The showDocument Method Is Optional

One serious caveat about this applet: The documentation for **AppletContext** describes **showDocument** as an optional method; browsers are *not* required to implement it. Different browsers implement in different ways, and some don't implement it at all.

The browser makers' reasoning is easy to divine. If you could jump around to various links via Java applets, using the browser simply as a base, the usefulness of the browser's fancy interface is called into question. In fact, the whole design of **showDocument** smells of the kind of corporate politics that often makes programming an exercise in nonsense-control. Why, if **showDocument**'s behavior is undefined, doesn't it at least return a value (instead of a void) or throw an exception to tell you whether it's going to do what you requested?

In Chapter 7, we'll take a more hands-on approach to Web trolling by recursively fetching HTML documents ourselves and searching them for patterns.

Reloading Applets

Most browsers support the notion of "reloading" an HTML page—clearing the browser window, rereading the HTML source from the http server, and redisplaying it in the browser window. This is useful, for example, where pages have cgi scripts embedded (for hit counting, for example) and the page needs to be downloaded for the script to run, or when a page currently in local cache is known to have changed back on the server. Unfortunately, there is no hard and fast rule as to just what it means to reload an applet.

Going strictly by the HTML example, what you might expect is that the applet gets stopped, unloaded (whatever that means), reread from the Web server, reloaded, and restarted. The simple applet in Listing 4.17, a modification of the singing applet shown in Listing 4.7, plays a different audio clip whenever **init**, **start**, **stop**, or **destroy** get called. Try it under your favorite browser and see what happens.

Listing 4.17 Testing the Reload Button

```
package chap4;

import java.awt.Graphics;
import java.applet.Applet;
import java.applet.AudioClip;
import java.net.*;

/** A simple app that plays tunes at the four different stops
along the applet execution trail - init, start, stop, and
destroy.  The parameters?
  StartClip
  StopClip
  DestroyClip
  InitClip

@version 1.0 12/20/1995
@author John Rodley
*/
public class ch4_fig13 extends Applet {
  String sAppletInfo;
  URL StartClipURL;
  URL StopClipURL;
  URL InitClipURL;
  URL DestroyClipURL;

  String pinfo[][] = {
    {''InitClip'', ''URL'',
          ''The URL of an audio clip played at init'' },
    {''StartClip'', ''URL'',
          ''The URL of an audio clip played at start'' },
    {''StopClip'', ''URL'',
          ''The URL of an audio clip played at stop'' },
    {''DestroyClip'', ''URL'',
          ''The URL of an audio clip played at destroy'' }
     };
```

```
/** Whenever a user visits our page, the browser calls this.
This is an override of an Applet method.. */
  public void start() {
    if( StartClipURL == null )
      return;
    play( StartClipURL );
  }

/** Whenever the user leaves our page, this gets called.
Another override of an Applet method.
*/
  public void stop() {
    if( StopClipURL == null )
      return;
    play( StopClipURL );
  }

/** Tell anyone who might inquire what we are.
@return A string describing this applet.
*/
  public String getAppletInfo() {
    return(''JRodley - Java Applets Book - Chapter 4 figure 13'');
    }

/** Tell anyone who might inquire how to use this applet.
@return A 2d string array describing the arguments to this
applet.
*/
  public String[][] getParameterInfo() {
    return pinfo;
    }

/** Play the destroy clip. */
  public void destroy() {
    if( DestroyClipURL != null )
      play( DestroyClipURL );
  }

/** Connect to the URLs specified by the StartClip and EndClip
parameters in the applet tag.
*/
  public void init() {
    try {
      InitClipURL = new URL(getParameter(''InitClip''));
      StartClipURL = new URL(getParameter(''StartClip''));
      StopClipURL = new URL(getParameter(''StopClip''));
```

```
      DestroyClipURL = new URL(getParameter(''DestroyClip''));
      } catch( MalformedURLException e )
          { System.out.println(''exception''+e );}
    if( InitClipURL != null )
      play( InitClipURL );
  }
}
```

For single-threaded applets, this is all very simple, but it gets much more complicated when applets start creating their own threads. It is very difficult, perhaps impossible, for a browser to shut down independent applet threads in an orderly fashion, mostly because there's no way to know what those threads are doing: Only the applet that created the threads knows that. Thus, it is incumbent upon the applet to shut down its own threads in **stop** and/or **destroy**.

Conclusion

Here, we've seen what applets look like, where they appear, and how they behave. We've shown how the interaction between the browser and its applets is strictly defined by the **Applet** class and the **AppletContext** interface. We've also discussed the methods available to an applet, the flow of execution, applet persistence, and the environment in which applets execute. And finally, with the Web-trolling applet, we've seen a little glimpse of how applets can enhance the functionality of browsers to the point of turning them into little more than Java platforms.